NASA Technical Memorandum 86354



# Efficient Implementation of Real-Time Programs Under the VAX/VMS Operating System

Sally C. Johnson

MAY 1985



NASA

NASA Technical Memorandum 86354

# Efficient Implementation of Real-Time Programs Under the VAX/VMS Operating System

Sally C. Johnson
*Langley Research Center*
*Hampton, Virginia*

NASA
National Aeronautics
and Space Administration
**Scientific and Technical Information Branch**
**1985**

## Summary

This paper is a user's guide for efficiently implementing real-time programs under the VAX/VMS[1] operating system. The techniques presented are for minimizing response times for a single real-time program executing on a dedicated VAX computer. A description of the basic operations needed to achieve real-time execution and techniques for optimizing efficiency are presented. A technique for decreasing the response time for accessing devices by mapping to the input/output (I/O) space and accessing device registers directly is discussed, and the resulting increase in performance is demonstrated by applying the technique to three of the devices available in the Langley Avionics Integration Research Lab (AIRLAB): the KW11-K dual programmable real-time clock, the Parallel Communications Link (PCL11-B) communication system, and the Datacom Synchronization Network. Each device is described, and methods for decreasing the access time for each device are then discussed. The examples show the use of the technique in three very different ways. These should provide sufficient background for applying the technique to other typical physical devices. The timing data included can be used to estimate the potential performance increase to be expected from applying the technique to other physical devices. The examples are presented in the PASCAL programming language; however, the technique can also be applied in other languages.

## Introduction

A real-time process must execute as efficiently as possible and avoid unnecessary delays in processing, usually because it is event or interrupt driven and time critical. This requires a dedicated computer and the required privileges to retain control of the system during real-time execution. The key to optimizing real-time performance lies in recognizing and avoiding conditions that can lead to processing delays and in accessing the resources available as efficiently as possible.

The VAX/VMS operating system contains many features designed to optimize processing and resource management for a wide range of competing interactive and batch processes. These features are unnecessary and can even be detrimental to a single, dedicated real-time process. System control is divided between processes via a complex priority scheme. When the process currently executing must wait for an event such as input/output (I/O), it may be swapped out of memory to make room for a currently executable process. Memory may be conserved by not copying all of a program into memory until it is accessed. All these features optimize the handling of competing processes in a multi-user environment; however, they are unneeded and should be disabled to optimize for a single, dedicated real-time process. Instructions for disabling these unnecessary features are included.

A real-time application requiring extensive communication with physical devices, such as clocks and communication links, will experience serious processing delays if these devices are not used efficiently. A highly efficient technique of mapping to the I/O space and accessing the device registers directly is therefore described. To illustrate the application of the technique, examples are included of different uses of the technique on three devices in the Langley Avionics Integration Research Lab (AIRLAB) for real-time applications: the KW11-K dual programmable real-time clock, the Parallel Communications Link (PCL11-B) communication system, and the Datacom Synchronization Network. The use of the technique on each device is discussed, and examples are given in the appendixes. The examples were chosen from a system for synchronizing four VAX computers by clock-value exchange in software, an application requiring short response times. The examples illustrate three different uses of the technique, which may be extended to other physical devices in a similar manner. Timing data are also included to show the performance increase realized from various uses of the technique. In addition, an extended-capability device driver available in AIRLAB for the KW11-K clock is discussed, and the parameters necessary for using this device driver are presented. Although the technique described is applicable regardless of programming language, all examples are presented in PASCAL.

Mapping to the I/O space to access physical device registers directly is a dangerous technique because an error could cause data stored on disk to be destroyed. Therefore, the use of this technique is not recommended unless the performance improvement is required for an application. To help the system designer make this choice, timing data are included to compare the performance improvement obtained from various uses of the technique with the same functions performed without use of the technique. Other techniques for using devices more efficiently are also recommended.

The first section describes how to perform real-time processing under the VAX/VMS operating system. Methods for using devices for real-time processing are discussed in the second section. The third

[1] VAX and VMS are trademarks of Digital Equipment Corporation.

section shows the response times required for VMS to handle some typical real-time requests. The fourth section describes a method for mapping to the I/O space to access device registers directly. The benefits as well as the dangers of this method are discussed in this section. The last three sections describe the use of the technique on three devices commonly used for real-time processing, namely, the KW11-K dual programmable real-time clock, the PCL11-B communication system, and the Synchronization Network. Each device is described, and examples are given for accessing each device more efficiently.

## Real-Time Processing Under VAX/VMS

A real-time program executes in three stages. Execution begins at a low-priority level. The executing program must instruct the operating system to disable operating system functions that would unnecessarily interrupt program execution and to raise the priority to a real-time level. The program is then executing in real time, and the time-critical processing may begin. After processing has completed, the program must instruct the operating system to lower the process priority to below a real-time level and to resume normal operating system functions.

The program communicates with the operating system via system service routines. These routines are explained fully in the *VAX/VMS System Services Reference Manual* (ref. 1). Each of the system service routines is a function returning a status code that indicates the success or failure of the operation requested. If the status code returned is odd, the operation was completed successfully. The following is an example of how to use the status code returned from a call to the $SETRWM system service to verify the success or to obtain a meaningful error message using the $GETMSG system service:

```
STATUS := $SETRWM(0);
   (* The system service is called as a
   function and returns a status code. *)
IF NOT ODD(STATUS) THEN BARF(STATUS);
   (* An odd status code indicates
   success. *)
PROCEDURE BARF(STATUS: INTEGER);
VAR
   ERRMSG: PACKED ARRAY[1..80] OF CHAR;
   MSGLEN: WORD;
   I: INTEGER;
BEGIN
   FOR I := 1 TO 80 DO ERRMSG[I] := ' ';
   $GETMSG(STATUS,MSGLEN,ERRMSG);
      (* The $GETMSG service interprets the
      status code and returns a system error
      message. *)
   WRITELN(ERRMSG);
END;
```

Before a PASCAL program can call a subroutine, even a system service routine, the complete specification of the routine parameters must be identified. To facilitate this, an environment file called STARLET containing the procedure specifications for all system service calls and system symbolic definitions, such as status codes and function codes, is available for PASCAL versions 2.0 and above. The environment must be "inherited" by including the following on the first line of a PASCAL program:

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM x;
```

The use of this feature is described fully in the *VAX-11 PASCAL User's Guide* (ref. 2). The system service routines are specified in the STARLET environment without the SYS prefix; for example, the system service SYS$GETMSG would be called $GETMSG.

Many of the system service routines may only be used by processes with appropriate privileges. The privileges that each process has are established by the system manager. Real-time users need more privileges than the average time-sharing user because of the functions they must perform which are specific to real-time processing.

### Before Real-Time Execution

Before real-time processing may begin, the operating system must be instructed to disable functions that would cause unnecessary delays during program execution and are unneeded by the program. All pages needed by the process should be locked in memory, and process swapping should be disabled before execution is raised to real-time priority. Many initialization procedures, such as starting clocks and establishing communication links, should be done before real-time processing begins.

***Locking pages in the working set.*** The entire program should be paged into the working set and locked-in before real-time execution begins to avoid paging delays during real-time processing. The beginning and ending addresses of the program to be locked-in are listed in the link map of the program, which may be obtained by linking the program with the qualifiers LINK/MAP/BRIEF.

The virtual-address space on the VAX computer is divided into two regions: the program region (virtual addresses 0 through $3FFFFFFF_{16}$) and the control region (virtual addresses $40000000_{16}$ through $7FFFFFFF_{16}$). The program region contains the program image currently executing, whereas the control

region contains the user stack and information maintained by the system on behalf of the process. Only pages in the program region may be locked-in.

The pages are locked into the working set using the $LKWSET system service with the beginning and ending addresses specified. For example, the system service request to lock in a program that starts at address $200_{16}$ and ends at address $7FF_{16}$ is

```
INADR[1]:=%X'0200';
INADR[2]:=%X'07FF';
$LKWSET(INADR,,);
```

The $LKWSET system service requires no privileges.

***Disabling swapping.*** Pages that are locked into the working set can still be swapped out if the process is idly waiting for some event. To avoid possible delays from waiting for the process to be copied back into memory, process swapping should be disabled. Swapping is disabled using the $SETSWM system service as follows:

```
$SETSWM(1);
```

The $SETSWM system service requires the PSWAPM privilege.

***Disabling the resource-wait mode.*** If the resource-wait mode is enabled (the default), then any process requesting an unavailable system resource is suspended until the resource becomes available. This can be disastrous for a real-time process. The resource may become available after a lengthy delay in processing. Even worse, since the real-time process is the only one executing, the resource may never become available and processing is suspended forever. Since the process has complete control of the system, no other processes may execute, and the only way to regain control of the "stuck" computer may be to shut it down and reboot. Therefore, the resource-wait mode should be disabled using the $SETRWM system service as follows:

```
$SETRWM(1);
```

The $SETRWM system service requires no privileges.

***Creating a time-out routine.*** A program executing in real-time priority has total control of the system. Therefore, if the program gets into a mode of looping endlessly or waiting forever for an event that never occurs, there is no way to regain control from the program without shutting down the computer and rebooting it. Therefore, it is recommended that a timer be set to execute a routine that halts the program after a specified amount of time has passed. The routine may also include output of error messages or other processing, but it need only include the following:

```
[ASYNCHRONOUS]PROCEDURE TOOLONG;
BEGIN
   (* System service calls to restore dis-
   abled operating system functions and
   lower process priority level should go
   here. *)
   HALT;
END;
```

The code for setting up a timer to execute this routine after 5 minutes is

```
TYPE
   QUAD = [QUAD,UNSAFE] RECORD
      LO:UNSIGNED; L1:INTEGER; END;
VAR
   TIMADR: QUAD;
BEGIN
   $BINTIM('0 0:5:0',TIMADR);
      (* The $BINTIM routine encodes
      the time value into system time
      format. *)
   $SETIMR(,TIMADR,TOOLONG);
      (* The $SETIMR routine sets up the
      routine TOOLONG to execute when the
      specified time has elapsed. *)
END;
```

***Raising priority to real time.*** Any process at priority 16 or higher is executing at real-time priority and will not be preempted by any non-real-time processes. Priority is set with the $SETPRI system service as follows:

```
$SETPRI(,,16,);
```

The $SETPRI system service requires the ALTPRI privilege.

Even after all these precautions, there are still occurrences that may delay processing. If the computer is connected to a DECnet[2] network, traffic over DECnet will not be able to pass the machine during real-time priority processing, but a short interruption will occur whenever the DECnet interface receives a communication. In AIRLAB, the computer may be disconnected from the DECnet network by a bypass switch located on the front of each computer. A brief hardware interrupt will occur every time a user

[2] DECnet is a trademark of Digital Equipment Corporation.

presses the ENTER key on any terminal connected to the computer. The operating system will still interrupt the process briefly every 30 msec to perform housekeeping functions. None of these interruptions cause serious delays; however, critical timing must allow for these brief delays. After the unnecessary operating system functions have been disabled and priority has been raised to a real-time level, time-critical processing may begin.

### Real-Time Execution

Many of the following features of the operating system may cause lengthy delays in processing, and their use is not recommended for efficient time-critical processing. Suspension or hibernation of a process surrenders control of the system, and lower priority processes may execute. When the real-time process becomes executable, a delay occurs while control is returned from the other process. Any subroutines to be executed during real-time execution should have local variables declared "static" so that critical processing time is not wasted during the allocation of storage. PASCAL I/O is slow and should be deferred until after time-critical processing has completed if possible. Also, the current version of PASCAL on the VAX computers in AIRLAB has a problem such that if interrupts or asynchronous processing should occur while PASCAL I/O is being processed, then unpredictable fatal execution errors may result.

### After Real-Time Execution

After time-critical processing has completed, the process should restore the operating system functions to their original states and lower priority to below real time before terminating. The system functions disabled may be restored with the following system service requests:

```
$SETPRI(,,4,);
   (* Lowers priority to previous level. *)
$SETRWM(0);
   (* Enables resource wait mode. *)
$SETSWM(0);
   (* Enables swapping. *)
$ULWSET(INADR,,);
   (* Unlocks pages in working set. *)
```

When possible, the output of diagnostic data should be delayed until after real-time processing has completed to avoid delays in time-critical processing and to avoid possible fatal execution errors if asynchronous processing should occur during PASCAL I/O.

## Using Devices for Real-Time Processing

Programs executing under the VAX/VMS operating system typically communicate with device drivers through system service routines to manipulate physical devices. Before a process can access a device driver, a communication link must be established between the process and the device. The $ASSIGN system service does this by assigning an I/O channel and returning the channel number by which the process must refer to the device. The process may then access the device driver using the $QIO or $QIOW system services, both of which have the following parameters:

| | |
|---|---|
| EFN | (optional) number of an event flag to be set when the specified operation has been performed |
| CHAN | I/O channel number assigned to the device |
| FUNC | code specifying the operation to be performed |
| IOSB | (optional) address for return of a status quadword indicating final completion status of the operation |
| ASTADR | (optional) address of the entry mask of an asynchronous system trap (AST) routine to be executed upon completion of the specified operation |
| ASTPRM | (optional) parameter to be passed to the AST routine |
| P1–P6 | (optional) device- and function-specific parameters |

The $QIO and $QIOW system services perform the same function; the only difference is in the return of control to the user process. The $QIO service returns control to the user process immediately upon setting up the device driver routine to execute. The $QIOW service retains system control until after the device driver has completed the requested operation.

The status code returned by the $QIO and $QIOW system services indicates the success of setting up the device driver to perform the desired operation. This status code is available when control is returned to the user process. The device driver may notify the process of completion of the operation performed by setting the specified event flag or by executing a specified AST routine. The executing program, when requesting certain system services, may specify one of its subroutines to be executed when the requested operation is completed. The AST routine must be declared [ASYNCHRONOUS] and may modify any global program variables that are declared

[VOLATILE]. (See ref. 3.) The device driver returns its own status code after the desired operation has been performed.

Before execution is raised to real-time level, initializing functions should be performed, such as assigning an I/O channel to each device, starting the clock, and establishing communication links. Each system service call to access a physical device introduces a delay of several milliseconds, which can significantly slow time-critical processing. Although this delay is acceptable for most real-time applications, the following section describes a technique for bypassing the system service call and device driver processing by directly writing to and reading from the device registers.

## VAX/VMS Response Time

Many real-time applications require crucial timing during execution, and the system designers of such applications should be aware of the response time of the VAX/VMS operating system. The VMS response time for a typical request is a few milliseconds. For illustration, the response times of various methods for VMS to notify a process that a clock interrupt has occurred are compared. These times are for a VAX-11/750 computer. Figure 1 shows that the elapsed time between a clock interrupt and the initiation of an AST routine of the process is approximately 1.4 msec. After a minimal AST routine (one PASCAL assignment statement) executes, control is returned to the main executing process approximately 1.5 msec after the clock interrupt, as shown in figure 2. If no AST routine executes, a process waiting for an event flag to be set upon a clock interrupt resumes execution approximately 1.4 msec after the clock interrupt, as shown in figure 3. These figures demonstrate that the VMS operating system requires approximately 1.5 msec to return control to an application program after a clock interrupt and that there is very little difference in timing between using an event flag and using an AST routine. The response time for the operating system to initiate execution of an AST routine upon clock interrupt is approximately the same; however, this response time has a considerably lower standard deviation.

The VMS operating system adds considerable overhead to real-time processing. The real-time applications designer must be aware of this overhead and take steps to avoid unnecessary delays in processing.

## Mapping to the I/O Space

A delay of several milliseconds is introduced when a process communicates with a physical device through a system service routine. However, since

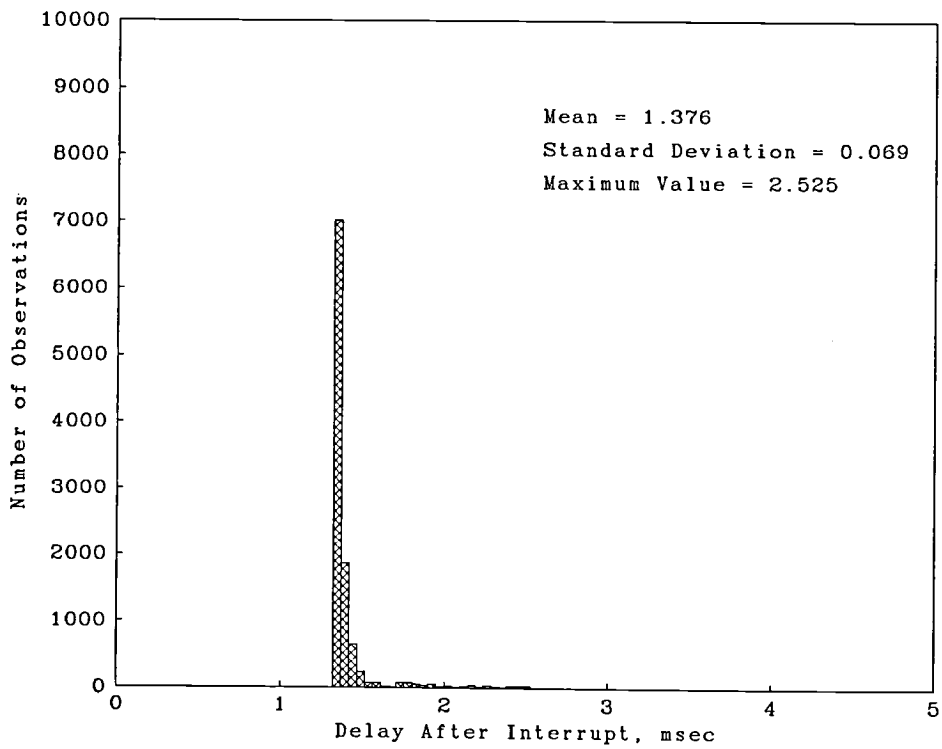


Figure 1. Histogram of elapsed time between clock interrupt and execution of AST routine to signal the interrupt.

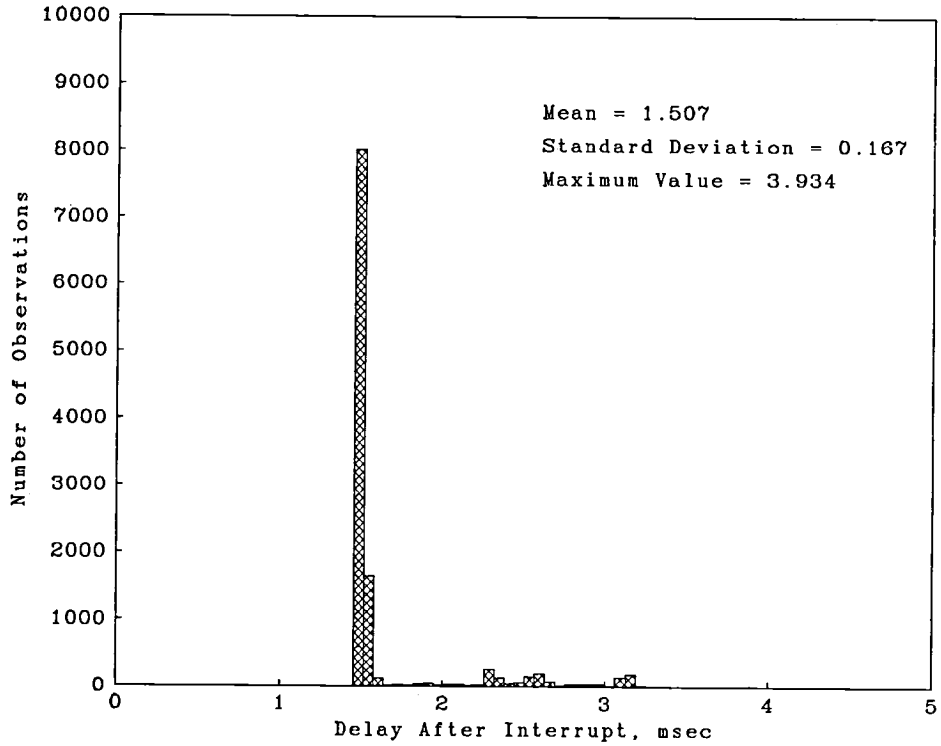


Figure 2. Histogram of elapsed time between clock interrupt and return of control to main program after short (one PASCAL statement) AST routine executes.

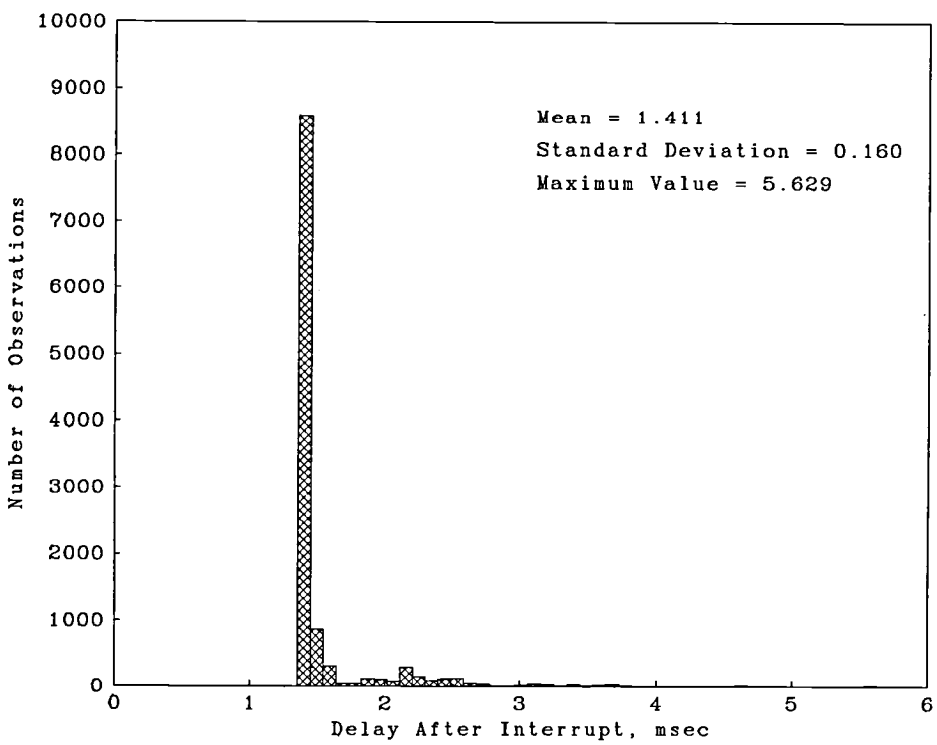


Figure 3. Histogram of elapsed time between clock interrupt and return of control to process waiting for event flag to be set upon clock interrupt.

the VAX has memory-mapped I/O, even a program written in a high-level language, such as PASCAL, can gain access to the device registers directly. This is accomplished by mapping a page of the program memory to the address of the page of I/O space containing the registers for a physical device. Storing a value in this mapped memory location by an assignment statement in the program is equivalent to loading a value into the physical device register. The registers can also be read by assignment statements. Thus, the program can use the device without using the device driver or system service routines. By avoiding system service routines and device driver processing, this technique dramatically reduces the delay involved in accessing a device from a few milliseconds to a few microseconds. For a more detailed discussion of this technique, see reference 4.

The register addresses for the PCL11-B communication system may be found in reference 5, and the register addresses for the KW11-K clock and the Synchronization Network devices are listed in reference 6. These devices may be located at different addresses on other systems. The addresses given in the manuals for each device are the addresses of the device registers on the UNIBUS.[3] The address of the UNIBUS adapter must be added to these addresses to obtain the physical device register addresses. All hardware devices are on the first UNIBUS adapter on the VAX-11/750 computers. The hardware addresses are different on the VAX-11/780; however, only the VAX-11/750 addresses are discussed herein.

Of a physical address, bits 9 to 23 are the virtual block number of the page and bits 0 to 8 are the byte offset within the page. Although the VAX computer is byte addressable, the device registers are words, so an array of words will be mapped to the I/O space. Therefore, the word offset within the page must be determined from the byte offset. The following is an example calculation for the address of the KW11-K clock counter register from its UNIBUS address found in reference 6:

The address of the KW11-K clock counter register is listed as $770430_8$. This UNIBUS address must be added to the first address on the first UNIBUS adapter to obtain the physical address of the register:

$$\begin{array}{rrl} 770430_8 = & 3F118 & \text{(hexadecimal)} \\ & \underline{+FC0000} & \text{(first UNIBUS address)} \\ & FFF118_{16} & \end{array}$$

Of the physical address, bits 9 to 23 are the virtual block number (VBN) of the page and bits 0 to 8 are the byte offset within the page:

$$FFF118_{16}$$

$$= \underbrace{1111\ 1111\ 1111\ 000}_{7FF8_{16}=\text{VBN}}\ \underbrace{1\ 0001\ 1000}_{118_{16}=\text{Byte offset within the page}}\ \text{Binary}$$

Because the array that will be indexed to access the physical register is an array of words, the word offset within the page must be calculated from the byte offset:

$$\begin{array}{rl} \text{Byte offset} = & 1\ 0001\ 1000 \\ \text{Shift right one bit} \rightarrow & \underline{\ \ 1000\ 1100} \\ & 8C_{16} = \text{Word offset within the page} \end{array}$$

A page of a PASCAL program memory may be mapped to this address by the following method. An array of words with a range of 0 to 255 is defined to be aligned on a page boundary. This array must have the [VOLATILE] attribute. The virtual address of this page is then mapped to the correct address of the I/O space by using the $CRMPSC system service. The array must not be locked in the working set. Examples of mapping to the I/O space of each of the three devices may be found in appendixes A, B, and C. The register addresses for each device may be found in table I.

## Dangers of Mapping to the I/O Space

Mapping to the I/O space is a dangerous technique and should only be attempted if the increase in performance warrants the risk. If the array is mapped to the wrong location in the I/O space or an incorrect word offset is used, other UNIBUS devices may be accessed by mistake causing unpredictable results. The possible risks include destroying data stored on disk. For this reason, the technique requires the privileges SHMEM and either PRMGBL or SYSGBL, and the technique is not recommended unless high efficiency is needed for an application.

Another technique for accessing a physical device more efficiently is to modify the device driver to make it more efficient for a specific application. Although this method is not as efficient as mapping to the I/O space because a system service request is still required, the inputs will be automatically checked by the system for validity when the device driver is called to avoid the risks inherent in uncontrolled writing to device registers.

For applications requiring faster response to device interrupts, the real-time program can also be connected to receive device interrupts directly. This

[3] UNIBUS is a trademark of Digital Equipment Corporation.

TABLE I. REGISTER ADDRESSES ON VAX-11/750 FOR COMMONLY USED DEVICES IN AIRLAB

| Device | VBN | Word offset | Register |
|---|---|---|---|
| KW11-K clock A | 7FF8 | 8A | Status Register |
| | | 8B | Preset/Buffer |
| | | 8C | Clock Counter |
| Synchronization Network | 7FF1 | 61 | Control Status Register |
| | | 62 | Status Register |
| | | 63 | Command Register |
| PCL11-B bus | 7FF4 | 40 | Transmitter Command Register (TCR) |
| | | 41 | Transmitter Status Register (TSR) |
| | | 42 | Transmitter Source Data Buffer (TSDB) |
| | | 43 | Transmitter Source Byte Count (TSBC) |
| | | 44 | Transmitter Source Byte Address (TSBA) |
| | | 45 | Transmitter Master/Maintenance (TMMR) |
| | | 46 | Transmitter Source Cyclic Redundancy Character (TSCRC) |
| | | 48 | Receiver Command Register (RCR) |
| | | 49 | Receiver Status Register (RSR) |
| | | 4A | Receiver Destination Data Buffer (RDDB) |
| | | 4B | Receiver Destination Byte Count (RDBC) |
| | | 4C | Receiver Destination Bus Address (RDBA) |
| | | 4D | Receiver Destination CRC (RDCRC) |

technique also avoids the dangers of accessing the device registers directly. The technique of connecting to interrupts is explained in the *VAX/VMS Real-Time User's Guide* (ref. 4).

## Benefits of Mapping to the I/O Space

In the following sections, the application of this technique to three devices is described in detail. Timing data are included to demonstrate the timing improvements realized. The examples come from a system for synchronizing four VAX computers through clock-value exchange in software. The devices manipulated are the KW11-K dual programmable real-time clock, the PCL11-B communication system, and the Synchronization Network. Although the use of this technique is only described for these three devices, other physical devices may be manipulated in a similar manner to achieve comparable performance increases.

The first example shows that even a simple device driver function requires a significant delay that can be avoided by accessing the device registers directly. The delay for reading the KW11-K clock can be reduced from a few milliseconds to a few microseconds. Reading the clock is a simple function in the device driver. A variable is checked to ensure that the clock is running, and then the clock counter register is read. Most of the time required for this function is the overhead for executing the `$QIO` system service routine, setting up the device driver to execute, and returning control to the application program. By reading the clock counter register directly from a single assignment statement in the application program, all this overhead is avoided. This shows that a simple function like reading the clock can be done very easily from within the application program, providing a dramatic increase in efficiency. The ability to decrease the overhead for reading the clock is especially important because the accuracy to which a clock can be used for timing is dependent on the accuracy with which it can be read.

The second example shows a considerable decrease in overhead when the technique is used to perform a more complex task. In this example, a one-word message is sent over the PCL11-B communication system without using the device driver at the sending or receiving computers. In this example, the application programs must perform considerably more complicated tasks than a simple assignment statement. However, the increase in efficiency for the function is still considerable. It should be noted that the communication delay measured is for sending a one-word message. The communication delay does not increase proportionally for longer messages, because the overhead for setting up to send a one-word message is approximately the same as for a longer message. Only the actual transmission delay and the delay for data retrieval are increased. Thus, the percentage of the communication delay that is operating system overhead is less for long messages than for short ones, so the performance increase of the technique would seem much less dramatic for long messages.

The third example shows a partial use of the technique. A pulse is sent over the Synchronization Network without using the device driver, but the receiver uses the device driver in the usual manner. As ex-

pected, the performance increase realized is much less than for the other examples. However, the technique still provides a noticeable increase in performance over complete dependence upon the device driver.

These examples show the use of the technique in three very different ways. These should provide sufficient background for applying the technique to most other typical physical devices. These examples also illustrate the performance increases to be expected from similar applications of the technique to other devices.

## The KW11-K Real-Time Clock

Accurate timing for real-time programs is provided by clock A of the KW11-K dual programmable clock. (See ref. 7.) This clock operates as a 16-bit up-counter. A negative value is loaded into the clock register from the Preset/Buffer. This value is incremented at a specified frequency, from 100 Hz to 1 MHz, until the register overflows (becomes equal to zero), signalling the end of an interval. The clock register is then reloaded from the Preset/Buffer and operation continues.

Included in the device-dependent parameters of the `$QIO` service for the KW11-K device driver are optional parameters for specifying an AST routine, a parameter to be passed to that routine, and an access mode for that routine to execute. Unlike the usual `$QIO` AST routine, which signals the user on completion of the task, this AST routine signals the user when the end of the next clock interval occurs.

A device driver written for AIRLAB by Datacom, Inc., and described in *Custom Software Documentation* (ref. 6) provides limited access to the following capabilities of the clock:

- starting the clock
- reading the current clock value
- stopping the clock
- causing an interrupt after a specified interval

To support real-time simulation in AIRLAB, the device driver was modified to enable use of the following features:

- generating repeated interval interrupts
- speeding up or slowing down the clock a specified amount for one interval without stopping the clock or interrupting its operation
- reading the current interval number

These functions of the modified device driver and how to use them are described in appendix D.

When the clock is operating at a 1-MHz rate, the clock value is incremented once every 1 $\mu$sec; thus, the clock is accurate to 0.001 msec. However, as shown in figure 4, there is a delay of 1.1 to 5.1 msec for the device driver to read the clock, so a clock read is only accurate to within a few milliseconds. This inaccuracy is unacceptable for many real-time applications. The following paragraph describes a method for reducing the delay required to read the clock.

Mapping to the I/O space and reading the clock counter register directly considerably reduces the delay required to read the KW11-K clock. As shown in figure 5, a clock read in this manner takes approximately 4 $\mu$sec and is accurate to within a few microseconds, a value which is much more acceptable than the several milliseconds required to read the clock using the device driver. These data were collected by repeatedly reading the clock directly 10 000 times and then determining the time between clock reads. Because the interval number is only accessible through the device driver, only the current clock counter value may be read in this manner. All other clock functions, such as starting and stopping the clock, can be performed using the device driver since the timing of these operations is not so critical. Appendix A contains the PASCAL code necessary for mapping to the I/O space containing the KW11-K clock registers and for reading the clock counter register directly.

## The PCL11-B Communication System

Although the DECnet interface is useful for initiating simultaneous execution of processes on multiple VAX computers in AIRLAB, rapid interprocess communication is provided by the PCL11-B communication system, a 16-bit parallel, time-division multiplexed bus. The *PCL11-B Driver User's Guide (XPDRIVER)* (ref. 8) describes how to establish communication links and send messages using the device driver. Each processor initializes itself and establishes a communication link with every other processor. A processor expecting to receive a message from another processor issues a `$QIO` request to its PCL11-B device driver identifying the expected receiver's address and an AST routine to be executed upon receipt of the message. To send a message, a processor issues a `$QIO` request specifying the message to be sent and the receiving node. If the receiving node has a read request pending, its AST routine executes and the message is delivered. Otherwise, the message is rejected by the receiving node.

Processing of the `$QIO` system service request and the device driver protocols adds a considerable delay to message transmission over the PCL11-B communication system. A histogram of one-word message delay times in figure 6 shows this delay to be from 5 to 11 msec. The vast majority of the delay results from the `$QIO` system service request and the

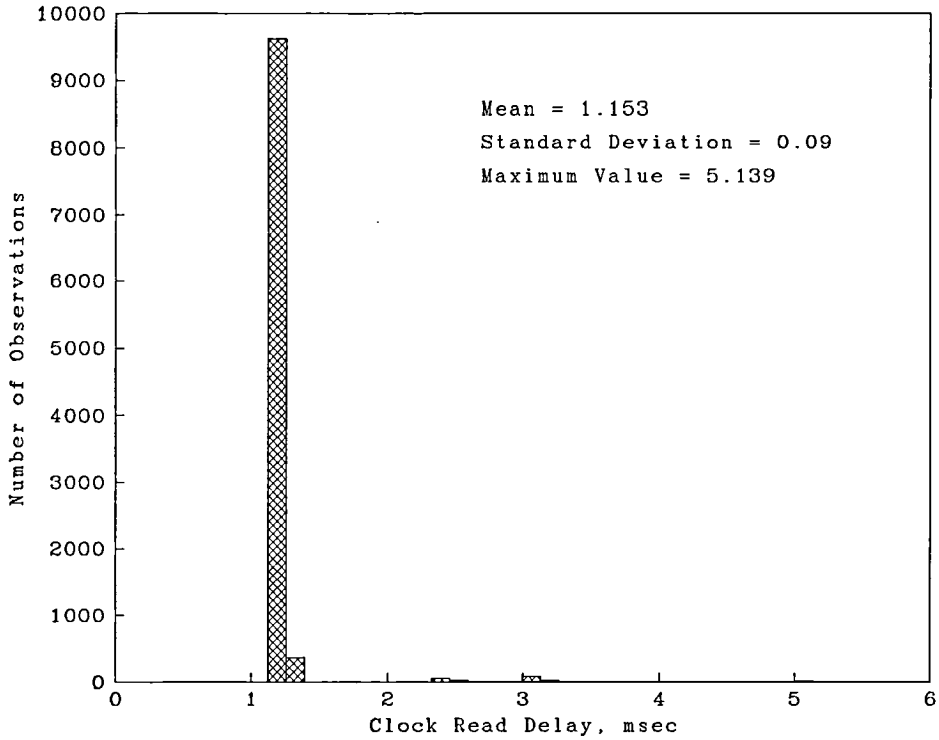


Figure 4. Histogram of delay for device driver to read KW11-K clock.

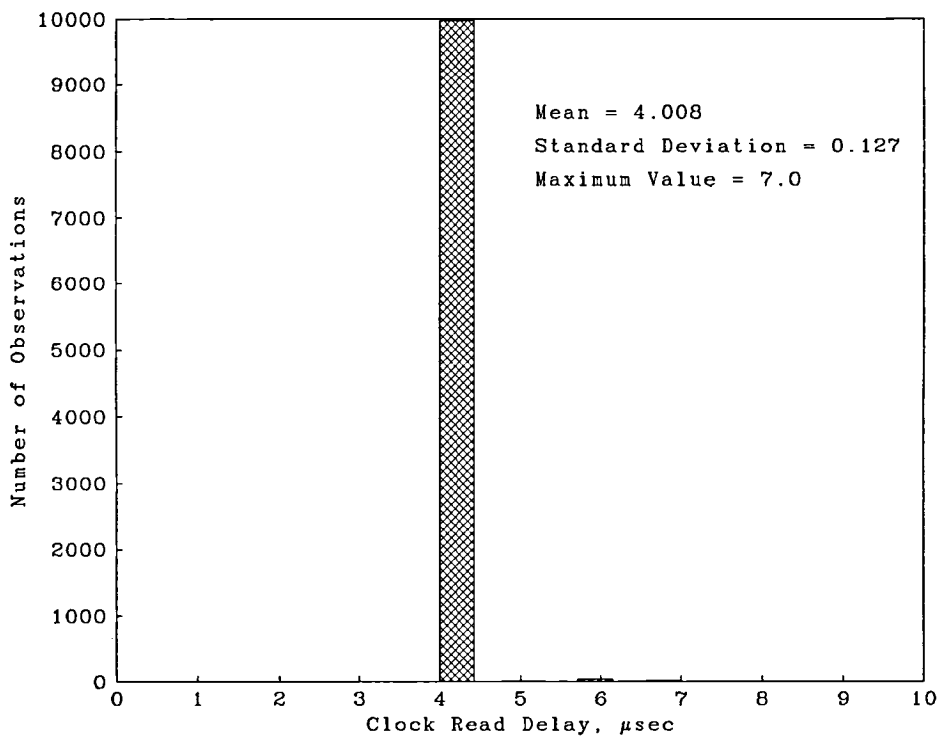


Figure 5. Histogram of delay for direct reading of KW11-K clock register.

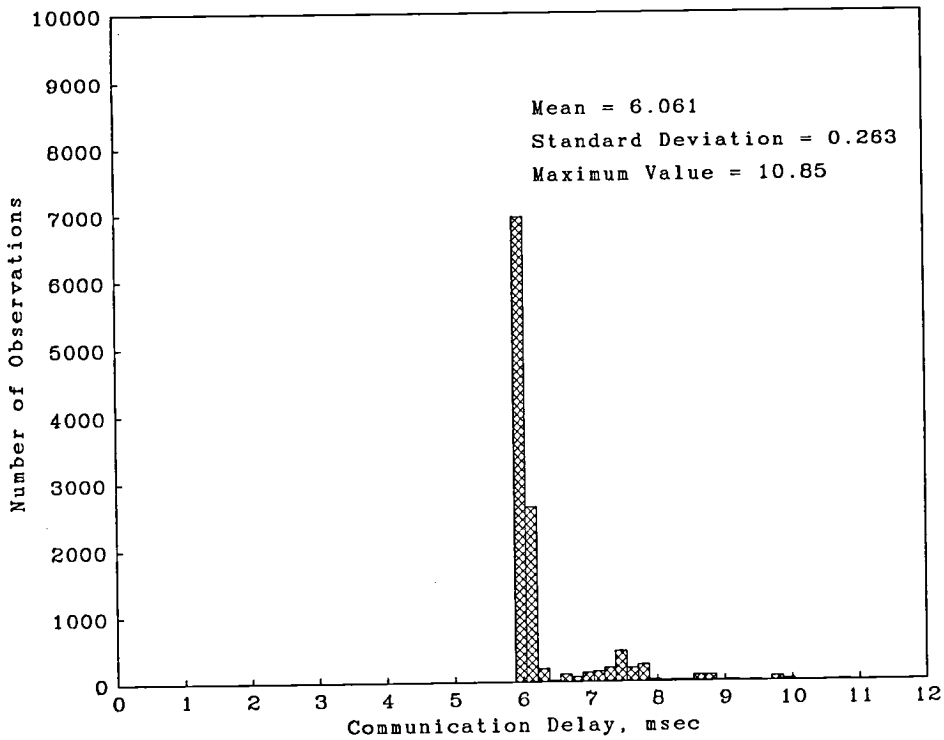


Figure 6. Histogram of delay for device driver to send one-word message over PCL11-B communication system.

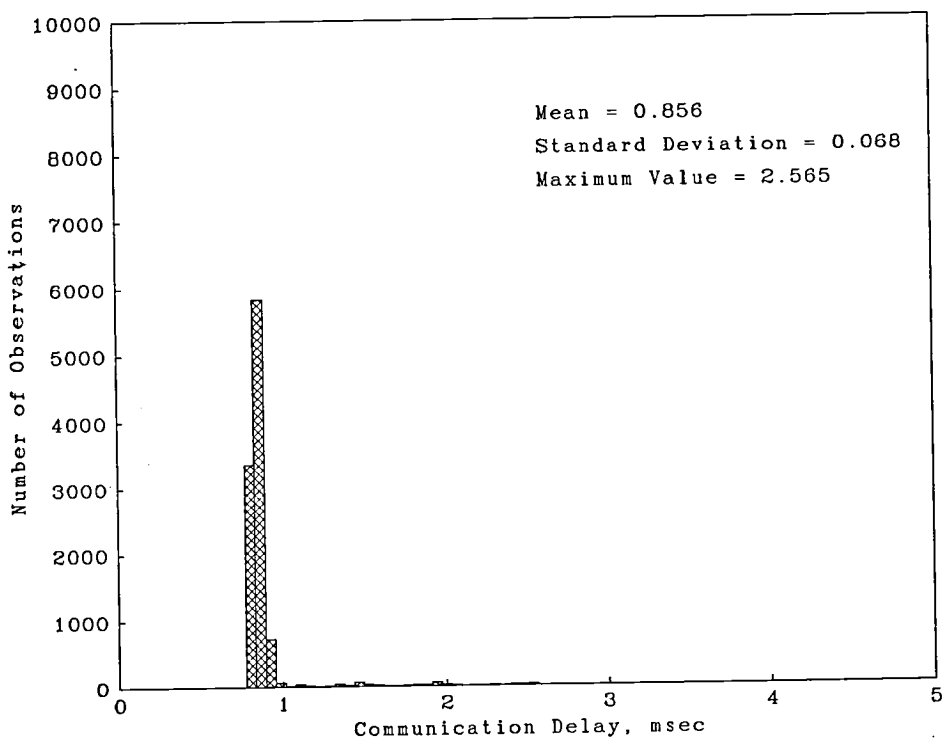


Figure 7. Histogram of delay for sending one-word PCL11-B message directly.

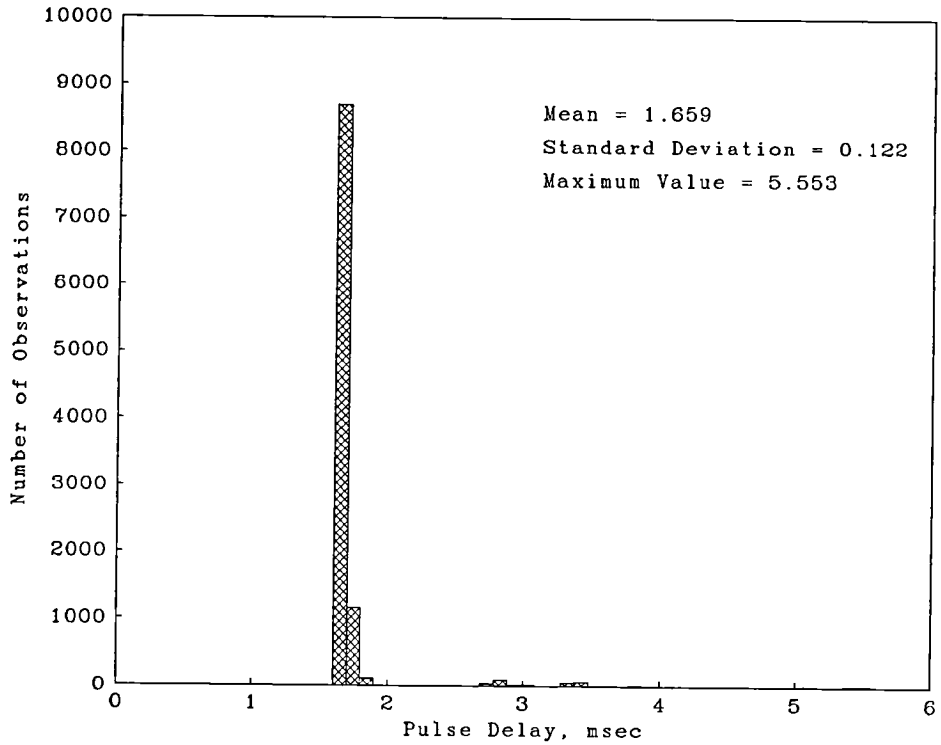


Figure 8. Histogram of delay for sending Synchronization Network pulse using device driver.

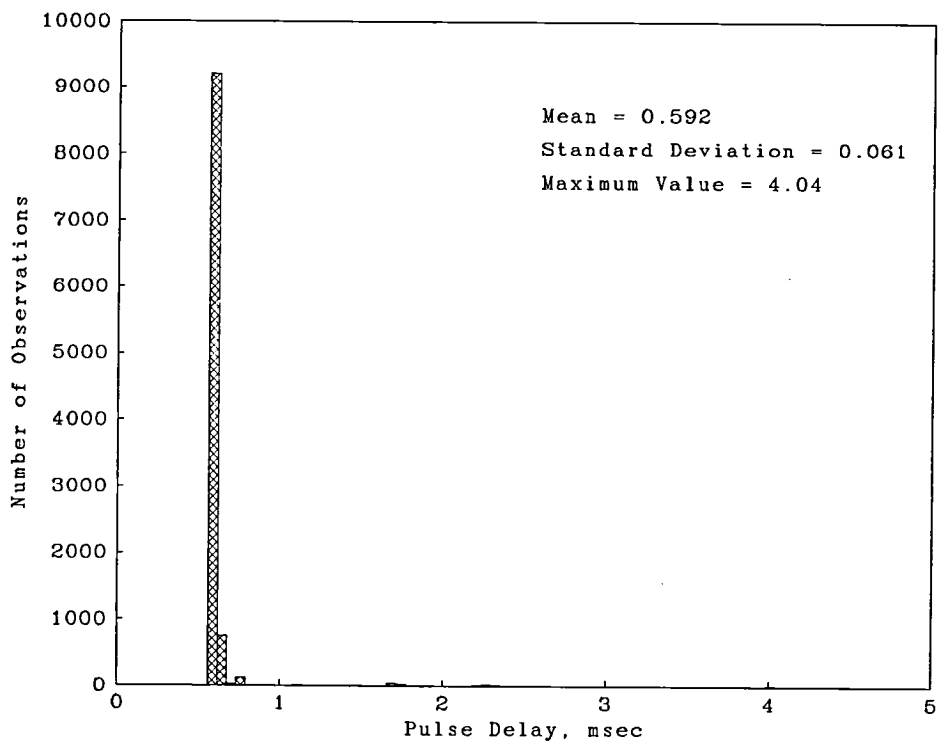


Figure 9. Histogram of delay for sending Synchronization Network pulse directly.

inefficient device driver code being designed to allow maximum generality of the device. For applications requiring shorter and less variable communication times, each process should map to the I/O space and messages should be exchanged by directly writing to and reading from the device registers. The sending node writes the message directly into the Transmitter Source Data Buffer register and then sets the bits to send the message to the receiver. The receiver must poll the device Receiver Status Register bits for the hardware interrupt to signal receipt of a message. The device driver may still be used to establish communication links, and then messages may be sent and received by accessing the device registers directly.

A method for mapping to the I/O space containing the PCL11-B device registers and for using the PCL without the device driver is illustrated by the example PASCAL routines in appendix B and is explained more fully in reference 5. As shown in figure 7, this technique reduces the communication delay to approximately 1 msec. The communication delay variance is also significantly reduced.

## The Synchronization Network

The Synchronization Network is a network for sending synchronization pulses between VAX computers and was developed for AIRLAB by Datacom, Inc. (See ref. 6.) With this system, a single pulse may be transmitted simultaneously to all the VAX computers in AIRLAB. The pulse may be sent by pressing the Manual Synchronization button or by issuing a `$QIO` write request to the Synchronization Network device driver from a program executing on one of the VAX computers. If an executing program requires notification upon receipt of a pulse from a specified sender or senders, then the program issues a read request to the Synchronization Network device driver. With this request, an AST routine may be specified to be executed upon receipt of the pulse. If no read requests are pending, then a pulse is ignored by the device driver.

As shown in figure 8, the delay for sending a synchronization pulse is approximately 1.5 msec. Synchronization pulses may be sent by mapping to the I/O space and writing to the device registers directly. Appendix C contains an explanation of how to send synchronization pulses without the device driver and gives an example of this technique. As shown in figure 9, this technique reduces the pulse delay to 0.6 msec with considerably less variation than with the device driver.

## Concluding Remarks

A description of techniques for writing efficient real-time programs under the VAX/VMS operating system was presented. Instructions for disabling unnecessary operating system functions that would cause needless processing delays and for raising the priority to a real-time level were included. Techniques to be avoided were also pointed out. To aid in planning time-critical programs, the response time of the VMS operating system was discussed.

A technique was presented for accessing physical devices more efficiently by mapping to the input/output space and accessing the physical-device registers directly. The dangers as well as the advantages of this technique were discussed. To illustrate the application of the technique, examples were included for using the technique on three different physical devices in the Langley Avionics Integration Research Lab. Each device was described, and suggestions for using each device efficiently were presented. The appendixes contain example PASCAL routines for using each of the devices in an efficient manner to illustrate most of the suggestions in the paper.

NASA Langley Research Center
Hampton, VA 23665
February 1, 1985

## Appendix A

## Reading the KW11-K Clock Register Directly

The following PASCAL routines map to the I/O page containing the clock registers and read the clock counter directly:

```
TYPE
   WORD = [WORD] -32768..32767;
CONST
   KWCLOCK = %X'8C';
VAR
   KWIOPAGE: [ALIGNED(9),VOLATILE]
     ARRAY[0..255] OF [VOLATILE] WORD;
      (* ALIGNED(9) specifies page
      alignment *)
   KWINADR,KWRETADR: ARRAY[1..2] OF
     [UNSAFE] UNSIGNED;
   MYTIME: WORD;
   STATUS: INTEGER;

PROCEDURE CLOCK_INIT;
VAR
   FLAGS: [UNSAFE] INTEGER;
   VBN: INTEGER;
BEGIN
   KWINADR[1] := ADDRESS(KWIOPAGE[0]);
   KWINADR[2] := KWINADR[1];
   VBN := %X'7FF8';
   FLAGS := SEC$M_PFNMAP;
   STATUS := $ULWSET(KWINADR,KWRETADR,);
   IF NOT ODD(STATUS) THEN BARF(STATUS);
   STATUS := $CRMPSC(KWINADR,KWRETADR,,
     FLAGS,,,,,1,VBN,,);
   IF NOT ODD(STATUS) THEN BARF(STATUS);
END;
```

The clock may now be directly read with the following:

```
MYTIME := KWIOPAGE[KWCLOCK];
```

## Appendix B

## Routines for Sending PCL11-B Messages Directly

The routines necessary for establishing communication links between two nodes and sending a message directly are given below. The constants and variables used are

```
CONST
    NMA$C_PCLI_PRO = %X'00000458';
    NMA$C_LINPR_MAS = %X'00000001';
    NMA$C_LINPR_SEC = %X'00000002';
    NMA$C_LINPR_NEU = %X'00000002';
    NMA$C_PCCI_TRI = %X'00000474';
    EFPCL = 2;
    TEF = 3;
    PCLTCR = %X'40';
    PCLTSR = %X'41';
    PCLTSDB = %X'42';
    PCLTSBC = %X'43';
    PCLTSBA = %X'44';
    PCLTMMR = %X'45';
    PCLTSCR = %X'46';
    PCLRCR = %X'48';
    PCLRSR = %X'49';
    PCLRDDB = %X'4A';
    PCLRDBA = %X'4B';
    PCLRDCRC = %X'4D';
    MAXWAIT = 10000;
TYPE
    WORD = [WORD] -32768..32767;
    QUAD = [QUAD,UNSAFE] RECORD
        LO:UNSIGNED; L1:INTEGER; END;
    IOSBTYPE = ARRAY[1..4] OF WORD;
    CBUF = RECORD
        LO:WORD;
        L1:INTEGER;
        END;
VAR
    MESSAGE: WORD;
    NDCHAN: [VOLATILE] ARRAY[1..10] OF WORD;
    CTRLCHAN: WORD;
    GOODREAD: [VOLATILE] ARRAY[1..10] OF
      [VOLATILE]BOOLEAN;
    BUFIN: [VOLATILE] ARRAY[1..10] OF
      [VOLATILE]WORD;
    IOSBR: [VOLATILE] IOSBTYPE;
    PCLIOPAGE: [ALIGNED(9),VOLATILE]
      ARRAY[0..255] OF [VOLATILE]WORD;
    PCLTEMP: [UNSAFE] WORD;
    PCLINADR,PCLRETADR: ARRAY [1..2] OF
      [UNSAFE]UNSIGNED;
```

The PCL_INIT routine maps the array PCLIOPAGE to the I/O space containing the PCL11-B device register addresses. The CONTROLLER_START routine starts the controller for this node. The NODE_CONNECT routine is then called to start every node in the system. Since a node may not be started until its controller is started, the attempts to start other nodes will fail until they start their controllers. After each node is started, messages may be sent and received, identifying each node by its assigned channel number.

```
PROCEDURE PCL_INIT;
VAR
    FLAGS: [UNSAFE]INTEGER;
    VBN: INTEGER;
BEGIN
    PCLINADR[1] := ADDRESS(PCLIOPAGE[0]);
    PCLINADR[2] := PCLINADR[1];
    VBN := %X'7FF4';
    FLAGS := UOR(SEC$M_PFNMAP,SEC$M_WRT);
    STATUS := $ULWSET(PCLINADR,PCLRETADR,);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
    STATUS := $CRMPSC(PCLINADR,PCLRETADR,,
      FLAGS,,,,,1,VBN,,);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
END; (* PCL_INIT *)

PROCEDURE CONTROLLER_START(THISNODE:
  INTEGER);
VAR
    CHARBUF: ARRAY[1..1] OF CBUF;
    QIO_FUNCT: [UNSAFE] INTEGER;
    IOSB: IOSBTYPE;
    NODENUM: INTEGER;
BEGIN
    STATUS := $ASSIGN('_XPA0:',CTRLCHAN);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
    CHARBUF[1].LO := NMA$C_PCLI_PRO;
    IF (THISNODE = 8) THEN CHARBUF[1].L1
      := NMA$C_LINPR_MAS
       ELSE IF (THISNODE = 6) THEN
         CHARBUF[1].L1 := NMA$C_LINPR_SEC
       ELSE CHARBUF[1].L1 := NMA$C_LINPR_NEU;

    (* One node should be chosen as master,
    one for secondary master, and all the
    other nodes must be neutral.*)

    QIO_FUNCT := UOR(IO$_SETMODE,UOR
      (IO$M_CTRL,IO$M_STARTUP));
    STATUS := $QIOW(,CTRLCHAN,QIO_FUNCT,%REF
      IOSB,,,,%DESCR CHARBUF);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
END; (* CONTROLLER_START *)

PROCEDURE NODE_CONNECT(NODENUM: INTEGER);
VAR
```

```
    CHARBUF: ARRAY[1..1] OF CBUF;
    QIO_FUNCT: [UNSAFE] INTEGER;
    IOSB: IOSBTYPE;
    TRIES,FLAG: INTEGER;
    CONNECTED: BOOLEAN;
    TIMADR: QUAD;
BEGIN
    STATUS := $ASSIGN('_XPAO:',NDCHAN
      [NODENUM]]);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
    CHARBUF[1].L0 := NMA$C_PCCI_TRI;
    CHARBUF[1].L1 := NODENUM;
    QIO_FUNCT := UOR(IO$_SETMODE,
      IO$M_STARTUP);
    FLAG := 0;
    TRIES := 0;
    CONNECTED := FALSE;
    WHILE (TRIES <= 100) AND NOT CONNECTED
      DO
        BEGIN TRIES := TRIES + 1;
        STATUS := $QIOW(,NDCHAN[NODENUM],
          QIOFUNCT,%REF IOSB,,,,%DESCR
          CHARBUF,FLAG);
        IF NOT ODD(STATUS) THEN BARF(STATUS);
        STATUS := $BINTIM('0 0:0:3',TIMADR);
        IF NOT ODD(STATUS) THEN BARF(STATUS);
        STATUS := $SETIMR(TEF,TIMADR);
        IF NOT ODD(STATUS) THEN BARF(STATUS);
        IF ODD(IOSB[1]) THEN CONNECTED := TRUE
          ELSE $WAITFR(TEF);
        END;
    IF NOT CONNECTED THEN
        BEGIN
        WRITELN(' NODE ',NODENUM:1,' NOT
          STARTED IN 100 TRIES');
        HALT;
        END;
END; (* NODE_CONNECT *)
```

Once the communication links are established by the CONTROLLER_START and NODE_CONNECT routines and a page of program memory is mapped to the I/O space containing the PCL11 device registers by the PCL_INIT routine, then messages may be sent by writing directly to the device registers using the following method:

The receiver must execute the READWAIT routine to poll the Receiver Status Register to recognize receipt of a message sent without the device driver as follows:

```
PROCEDURE READWAIT(NODENUM,MAXCOUNT:
  INTEGER):BOOLEAN;
VAR
    ICOUNT: [STATIC] INTEGER;
    IRCV: [STATIC] INTEGER;
BEGIN
    ICOUNT := 0;
    PCLIOPAGE[PCLRCR] := %X'0002';
    PCLIOPAGE[PCLRCR] := %X'2000';
    READWAIT := FALSE;
    IRCV := 0;
    REPEAT
        ICOUNT := ICOUNT + 1;
        PCLTEMP := PCLIOPAGE[PCLRSR];
        IF (UAND(PCLTEMP,%X'0100')
          = %X'0100') THEN IRCV := 1
        ELSE IF (UAND(PCLTEMP,%X'0080')
          = %X'0080') THEN IRCV = 1;
        UNTIL ((IRCV > 0) OR (ICOUNT
          > MAXCOUNT));
    IF (IRCV > 0) THEN READWAIT := TRUE;
    BUFIN[NODENUM] := PCLIOPAGE[PCLRDDB];
END; (* READWAIT *)
```

The sending node may write a one-word message directly into the Transmitter Source Data Buffer register and set the bits to transmit the message to the specified receiver as follows:

```
PROCEDURE WRITEPCL(NODENUM: INTEGER; BUF:
  WORD);
BEGIN
    PCLIOPAGE[PCLTCR] := %X'0002';
    PCLIOPAGE[PCLTSBC] := %X'FFFE';
    PCLTEMP := PCLIOPAGE[PCLTCR];
    IF (NODENUM = 6) THEN PCLTEMP := UOR
      (PCLTEMP,%X'0600')
       ELSE IF (NODENUM = 8) THEN PCLTEMP
         := UOR(PCLTEMP,%X'0800');
    PCLIOPAGE[PCLTCR] := PCLTEMP;
    PCLIOPAGE[PCLTSDB] := BUF;
    PCLTEMP := PCLIOPAGE[PCLTCR];
    PCLTEMP := UOR(PCLTEMP,%X'2000');
    PCLIOPAGE[PCLTCR] := PCLTEMP;
    PCLTEMP := PCLIOPAGE[PCLTSR];
    PCLTEMP := UAND(PCLTEMP,%X'FF7F');
    PCLIOPAGE[PCLTSR] := PCLTEMP;
    PCLTEMP := PCLIOPAGE[PCLTCR];
    PCLTEMP := UOR(PCLTEMP,%X'0001');
    PCLIOPAGE[PCLTCR] := PCLTEMP;
END; (* WRITEPCL *)
```

# Appendix C

## Routines for Sending Synchronization Pulses Directly

The routines necessary for setting up a node to receive a synchronization pulse and for sending a pulse by writing directly to the device registers use the following constants and variables:

```
CONST
    SYNCNET_CSR = %X'61';
    SYNCNET_COMMAND = %X'63';
    COMMAND = %X'40';
TYPE
    WORD = [WORD] -32768..32767;
VAR
    NSIOPAGE: [ALIGNED(9)] ARRAY[0..255] OF
      [VOLATILE] WORD;
    NSADR,NSRETADR: ARRAY[1..2] OF [UNSAFE]
      UNSIGNED;
    SYNCFLAG: [VOLATILE] BOOLEAN;
    NSTEMP: [VOLATILE,UNSAFE] WORD;
    SYNCCHAN: WORD;
```

Each node must map an array to the page of I/O space containing the Synchronization Network device register addressed as follows:

```
PROCEDURE SN_INIT;
VAR
    FLAGS: [UNSAFE] INTEGER;
    VBN: INTEGER;
BEGIN
    NSINADR[1] := ADDRESS(NSIOPAGE[0]);
    NSINADR[2] := NSINADR[1];
    VBN := %X'7FF1';
    FLAGS := UOR(SEC$M_PFNMAP,SEC$M_WRT);
    STATUS := $ULWSET(NSINADR,NSRETADR,);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
    STATUS := $CRMPSC(NSINADR,NSRETADR,,
      FLAGS,,,,,1,VBN,,);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
    STATUS := $ASSIGN(DEVNAM := '_SYA0:',
      CHAN := SYNCCHAN);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
END;
```

The RECVSYNC routine sets up the node to receive a synchronization pulse, which may be sent by another process writing to the device registers directly or issuing a write $QIO to the device or by pressing the Manual Synchronization button in AIRLAB. The SYNCAST routine will be executed when the first pulse is received.

```
[ASYNCHRONOUS] PROCEDURE SYNCAST;
BEGIN
    SYNCFLAG := TRUE;
END;

PROCEDURE RECVSYNC;
VAR
    MODEMASK: WORD;
    MODE: INTEGER;
BEGIN
    MODEMASK := %X'03FF'; (* Mask to receive
      from any other process *)
    MODE := 1;
    STATUS := $QIOW(,SYNCCHAN,IO$_SETMODE,
      P1 := MODEMASK,P2 := MODE);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
    SYNCFLAG := FALSE;
    STATUS := $QIO(,SYNCCHAN,IO$_READVBLK,
      P1 := %IMMED SYNCAST);
    IF NOT ODD(STATUS) THEN BARF(STATUS);
    IF (SYNCFLAG) THEN
        BEGIN
            (* The Synchronization Network
            is set up such that if an unex-
            pected synchronization pulse is
            received, it is remembered and
            the next read request will imme-
            diately be successful; however,
            for many applications this would
            be incorrect, and this check
            should be included. *)
        SYNCFLAG := FALSE;
        STATUS := $QIO(,SYNCCHAN,IO$_READVBLK,
          P1 := %IMMED SYNCAST);
        IF NOT ODD(STATUS) THEN BARF(STATUS);
        END;
END;
```

The routine SENDSYNC, for sending a synchronization pulse to every computer by writing directly to the synchronization device registers, is simple, as follows:

```
PROCEDURE SENDSYNC;
BEGIN
    NSTEMP := UOR(NSIOPAGE[SYNCNET_CSR],
      COMMAND);
    NSIOPAGE[SYNCNET_CSR] := NSTEMP;
    NSIOPAGE[SYNCNET_COMMAND] := %X'0000';
END;
```

## Appendix D

## An Extended-Capability Device Driver for the KW11-K Clock

The KW11-K device driver written by Datacom, Inc., was modified to extend its capabilities for real-time processing. The added functions of this device driver and the parameters for using them are discussed in this appendix.

Status codes are returned from the device driver in the IOSB parameter to the $QIO system service. The following status codes may be returned in the first longword of the IOSB:

| | |
|---|---|
| SS$_ILLSEQOP | an attempt was made to read the clock before it was started |
| SS$_IVTIME | an invalid value was specified for the Preset/Buffer |
| SS$_ASTFLT | an error occurred in queuing up the AST routine |
| SS$_CANCEL | successful completion of a request to stop the clock |
| SS$_NORMAL | successful completion of any other function |

The second longword of the IOSB is used by the clock read function to return the clock counter value and interval number and is not used by the other functions.

The modified KW11-K clock device driver accepts only READ and WRITE function codes. The operation for reading the clock counter is the only READ function and may be specified by the $QIO function code parameter IO$_READVBLK. All the remaining operations are WRITE functions and may be specified by the IO$_WRITEVBLK function code.

The WRITE function is used to perform four different functions in the modified device driver. The four functions are differentiated as follows:

| | |
|---|---|
| P4 = 0, P5 = 0 | stop the clock |
| P4 = 0, P5 ≠ 0 | change Preset/Buffer only |
| P4 ≠ 0, P5 = 0 | set up for interrupt only |
| P4 ≠ 0, P5 ≠ 0 | start the clock |

The parameters for each of the functions of the modified KW11-K clock device driver are given below. Example routines for using each of these functions may be found in appendix E. Further information on the modified clock driver is available on the AIRLAB Data Management System (ADAMS).

### Starting the Clock

The KW11-K clock may be started by a WRITE function with the following device-specific parameters:

| | |
|---|---|
| P1 | optional address of AST to be delivered at interval end |
| P2 | optional AST parameter |
| P3 | optional AST access mode |
| P4 | clock Control Status Register (CSR) (see ref. 7) |
| P5 | positive integer, equals number of clock ticks to be in each interval |

The *KW11-K Dual Programmable Real Time Clock User Manual* (ref. 7) should be consulted for the format of the clock CSR parameter and for further information about the operation of the KW11-K clock. The clock start function may be used to generate an interrupt at the end of the first interval by setting the correct bit in the clock CSR. An AST to execute upon that interrupt may be specified in parameter P1, and a parameter to be passed to it may be specified in P2.

### Queuing an AST or Event Flag for the Next Interrupt

Without disturbing the operation of the clock, this WRITE function may be used to signal the next clock interrupt by executing an AST routine or by setting an event flag. The device-specific parameters are as follows:

| | |
|---|---|
| P1 | optional address of AST to be delivered at interval end |
| P2 | optional AST parameter |
| P3 | optional AST access mode |
| P4 | positive integer |
| P5 | zero |

This function may be used to generate repeated interval interrupts in the following manner. A $QIO request is made to the clock device driver as before to start the clock operating, with the clock CSR set for repeated interval mode operation with interrupts enabled and with an AST routine specified to execute at the end of the first interval. After the AST routine executes, a $QIO request is issued to the device driver with this function to queue an AST to execute at the end of the next interval without interrupting clock operation.

By using this function at the beginning of every subsequent interval, the AST routine will execute at the end of every interval.

### Changing the Clock Preset/Buffer

The WRITE function for changing the value in the clock Preset/Buffer without disturbing the current operation of the clock requires the following device-specific parameters:

| | |
|---|---|
| P1 | optional address of AST to be delivered at interval end |
| P2 | optional AST parameter |
| P3 | optional AST access mode |
| P4 | zero |
| P5 | positive integer, equals new interval length |

This function changes the value in the Preset/Buffer without stopping or disturbing the operation of the clock. If the clock is running in repeated interval mode, the current interval length will remain unchanged, but all subsequent intervals will be the length specified in P5. In a multiprocessor system, this function may be used to speed up a lagging clock to synchronize with other processors' clocks by the following method. A $QIO call is made to the device driver to change the value in the Preset/Buffer to a slightly larger (less negative) value. At the end of the current interval, this value is automatically loaded into the clock register. This causes the next interval end to occur slightly earlier, so the end of this interval will coincide with the other processors' clocks. During the shortened interval, this function may be called with the original interval length to restore the Preset/Buffer. The clock resumes normal operation but is synchronized with the other processors' clocks. Note that control is returned to the calling program when the value has been successfully written to the Preset/Buffer, so another call to this function during the same interval could overwrite the value and cancel out the effect of the first call.

### Stopping the Clock

The function for stopping the clock is a WRITE function with parameters P4 and P5 both equal to zero. The device-specific parameters are thus as follows:

| | |
|---|---|
| P1 | zero |
| P2 | zero |
| P3 | zero |
| P4 | zero |
| P5 | zero |

### Reading the Clock

The function for reading the clock counter is a READ function with no device-specific parameters. The value in the clock counter and the interval number since clock operation was started are returned in the IOSB. Word three of the IOSB contains the counter value, and word four contains the interval number. Calculation of an absolute time value from the interval number and the counter value is shown in appendix E. The interval number is accessible only through a $QIO or $QIOW call to the device driver. Since the value is returned in the IOSB by the device driver, it cannot be read until the device driver has completed processing (i.e., when the $QIOW returns or the $QIO sets an event flag or delivers an AST routine). An attempt to read the clock when it is not operating will fail and will return a device status code of SS$_ILLSEQOP in the IOSB.

# Appendix E

## Routines for Using the Extended Capability KW11-K Device Driver

The following are example routines for each of the functions available for the extended-capability device driver for the KW11-K clock. The variables used are

```
TYPE
   WORD: [WORD] -32768..32767;
   QUAD: [QUAD,UNSAFE] RECORD
      L0: UNSIGNED;
      L1: INTEGER;
      END;
   TIMEQUAD: RECORD CASE INTEGER OF
      0:(Q:QUAD);
      1:(W:ARRAY[1..4] OF WORD;
      END;
VAR
   READKWA: TIMEQUAD;
   STATUS: INTEGER;
   CLOCKIOSB: QUAD;
   KWACHAN: WORD;
   INTFLAG: [VOLATILE] BOOLEAN;
   TIME: INTEGER;
```

The following routines may be used to generate repeated interval interrupts. The CLOCK_START routine starts the clock in repeated interval mode with intervals of 30 msec and queues the INTAST routine to be executed upon the first interrupt. The flag INTFLAG may be used like an event flag, but it may be read without a time-consuming system service request. When INTFLAG becomes true, the first interval has ended. The routine SETUP_INT should be called to queue the INTAST routine some time during the second interval and during each subsequent interval.

```
[ASYNCHRONOUS] PROCEDURE INTAST;
BEGIN
   INTFLAG := TRUE;
END;

PROCEDURE CLOCK_START;
BEGIN
   INTFLAG := FALSE;
   STATUS := $QIO(,KWACHAN,IO$_WRITEVBLK,
     CLOCKIOSB,,,%IMMED INTAST,,,%X'0143',
     30000);
   IF NOT ODD(STATUS) THEN BARF(STATUS);
END;

PROCEDURE SETUP_INT;
BEGIN
   INTFLAG := FALSE;
   STATUS := $QIO(,KWACHAN,IO$_WRITEVBLK,
     CLOCKIOSB,,,%IMMED INTAST,,,
     %X'0143',0);
END;
```

The CLOCK_CHANGE routine may be used to change the clock Preset/Buffer without disturbing the operation of the clock. The value NEWDELAY will be loaded into the clock counter at the end of the current interval. To change the length of only one interval, this routine must be called again with the original count value some time during the next interval.

```
PROCEDURE CLOCK_CHANGE(NEWDELAY: INTEGER);
BEGIN
   STATUS := $QIO(,KWACHAN,IO$_WRITEVBLK,
     CLOCKIOSB,,,,,,0,NEWDELAY);
   IF NOT ODD(STATUS) THEN BARF(STATUS);
END;
```

The clock operation may be stopped by the following routine:

```
PROCEDURE CLOCK_STOP;
BEGIN
   STATUS := $QIO(,KWACHAN,IO$_WRITEVBLK,
     CLOCKIOSB,,,,,,0,0);
   IF NOT ODD(STATUS) THEN BARF(STATUS);
END;
```

The clock counter may be read directly by the method described in appendix A, or the following method may be used to determine from the interval number and current interval size an absolute time from the time the clock was started.

```
PROCEDURE CLOCK_READ;
BEGIN
   STATUS := $QIOW(,KWACHAN,IO$_READVBLK,
     READKWA.Q);
   IF NOT ODD(STATUS) THEN BARF(STATUS);
   TIME := (KWACOUNT * READKWA.W[4])
     + (KWACOUNT + READKWA.W[3]);
END;
```

## References


1. *VAX/VMS System Services Reference Manual.* Order No. AA-D018C-TE, Digital Equipment Corp., May 1982.
2. *VAX-11 PASCAL User's Guide.* Order No. AA-H485C-TE, Digital Equipment Corp., Oct. 1982.
3. *VAX-11 PASCAL Language Reference Manual.* Order No. AA-H484C-TE, Digital Equipment Corp., Oct. 1982.
4. *VAX/VMS Real-Time User's Guide.* Order No. AA-H784B-TE, Digital Equipment Corp., May 1982.
5. *PCL11-B Parallel Communication Link Differential TDM Bus.* Doc. No. YC-A20TC-00, Rev. B, Digital Equipment Corp., c.1982.
6. *Custom Software Documentation, Volume 1.* Datacom, Inc.
7. *KW11-K Dual Programmable Real Time Clock User Manual.* EK-KW11-K-OP-001, Digital Equipment Corp., c.1976.
8. *PCL11-B Driver User's Guide (XPDRIVER).* Digital Equipment Corp., Jan. 1983.

| 1. Report No. | 2. Government Accession No. | 3. Recipient's Catalog No. |
|---|---|---|
| NASA TM-86354 | | |

| 4. Title and Subtitle | 5. Report Date |
|---|---|
| Efficient Implementation of Real-Time Programs Under the VAX/VMS Operating System | May 1985 |
| | 6. Performing Organization Code: 505-34-13-32 |
| 7. Author(s): Sally C. Johnson | 8. Performing Organization Report No.: L-15900 |
| 9. Performing Organization Name and Address: NASA Langley Research Center, Hampton, VA 23665 | 10. Work Unit No. |
| | 11. Contract or Grant No. |
| 12. Sponsoring Agency Name and Address: National Aeronautics and Space Administration, Washington, DC 20546 | 13. Type of Report and Period Covered: Technical Memorandum |
| | 14. Sponsoring Agency Code |

15. Supplementary Notes

16. Abstract

Techniques for writing efficient real-time programs under the VAX/VMS operating system are presented. Basic operations are presented for executing at real-time priority and for avoiding needless processing delays. A highly efficient technique for accessing physical devices by mapping to the input/output space and accessing the device registers directly is described. To illlustrate the application of the technique, examples are included of different uses of the technique on three devices in the Langley Avionics Integration Research Lab (AIRLAB): the KW11-K dual programmable real-time clock, the Parallel Communications Link (PCL11-B) communication system, and the Datacom Synchronization Network. Timing data are included to demonstrate the performance improvements realized with these applications of the technique.

17. Key Words (Suggested by Authors(s))
Real time
VAX/VMS operating system
Programming techniques

18. Distribution Statement
Unclassified—Unlimited

Subject Category 61

| 19. Security Classif.(of this report) | 20. Security Classif.(of this page) | 21. No. of Pages | 22. Price |
|---|---|---|---|
| Unclassified | Unclassified | 22 | A02 |

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1985